Beating Floating Point at its Own Game: Posit Arithmetic

***John L. Gustafson*[1]*, Isaac Yonemoto*[2]**

A new data type called a *posit* is designed as a direct drop-in replacement for IEEE Standard 754 floating-point numbers (floats). Unlike earlier forms of universal number (unum) arithmetic, posits do not require interval arithmetic or variable size operands; like floats, they round if an answer is inexact. However, they provide compelling advantages over floats, including larger dynamic range, higher accuracy, better closure, bitwise identical results across systems, simpler hardware, and simpler exception handling. Posits never overflow to infinity or underflow to zero, and "Not-a-Number" (NaN) indicates an action instead of a bit pattern. A posit processing unit takes less circuitry than an IEEE float FPU. With lower power use and smaller silicon footprint, the posit operations per second (POPS) supported by a chip can be significantly higher than the FLOPS using similar hardware resources. GPU accelerators and Deep Learning processors, in particular, can do more per watt and per dollar with posits, yet deliver superior answer quality.

A comprehensive series of benchmarks compares floats and posits for decimals of accuracy produced for a set precision. Low precision posits provide a better solution than "approximate computing" methods that try to tolerate decreased answer quality. High precision posits provide more correct decimals than floats of the same size; in some cases, *a 32-bit posit may safely replace a 64-bit float.* In other words, posits beat floats at their own game.

Keywords: computer arithmetic, energy-efficient computing, floating point, posits, LINPACK, linear algebra, neural networks, unum computing, valid arithmetic.

1. Background: Type I and Type II Unums

The ***unum*** (***u***niversal ***num***ber) arithmetic framework has several forms. The original "Type I" unum is a superset of IEEE 754 Standard floating-point format [2, 7]; it uses a "ubit" at the end of the fraction to indicate whether a real number is an exact float or lies in the open interval between adjacent floats. While the sign, exponent, and fraction bit fields take their definition from IEEE 754, the exponent and fraction field lengths vary automatically, from a single bit up to some maximum set by the user. Type I unums provide a compact way to express *interval arithmetic*, but their variable length demands extra management. They can duplicate IEEE float behavior, via an explicit rounding function.

The "Type II" unum [4] abandons compatibility with IEEE floats, permitting a clean, mathematical design based on the *projective reals*. The key observation is that signed (two's complement) integers map elegantly to the projective reals, with the same wraparound of positive numbers to negative numbers, and the same ordering. To quote William Kahan [5]:

> "They typically save storage space because what you're manipulating are not the numbers, but *pointers* to the values. And so, it's possible to run this arithmetic very, very fast."

The structure for 5-bit Type II unums is shown in fig. 1. With n bits per unum, the "u-lattice" populates the upper right quadrant of the circle with an ordered set of $2^{n-3} - 1$ real numbers x_i (not necessarily rational). The upper left quadrant has the negatives of the x_i, a reflection about the vertical axis. The lower half of the circle holds *reciprocals* of numbers in the top half, a reflection about the horizontal axis, making × and ÷ operations as symmetrical as + and −. As with Type I, Type II unums ending in `1` (the ubit) represent the open interval between adjacent exact points, the unums for which end in `0`.

[1]A*STAR Computational Resources Centre and National University of Singapore (joint appointment), Singapore
[2]Interplanetary Robot and Electric Brain Company, Saratoga, California USA

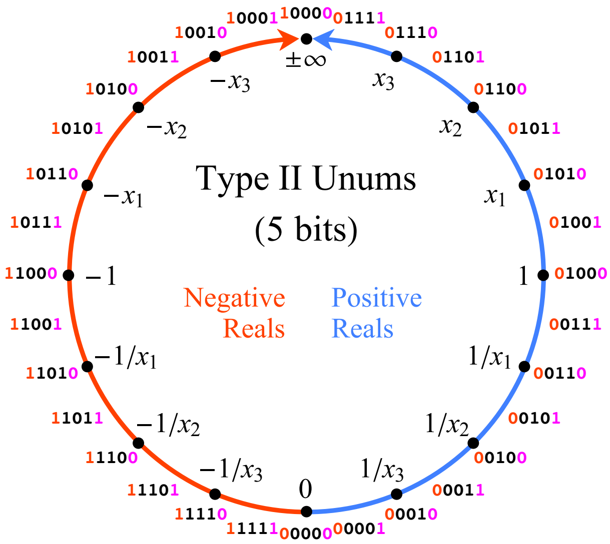

Figure 1. Projective real number line mapped to 4-bit two's complement integers

Type II unums have many ideal mathematical properties, but rely on **table look-up** for most operations. If they have n bits of precision, there are (in the worst case) 2^{2n} table entries for 2-argument functions, though symmetries and other tricks usually reduce that to a more manageable size. Table size limits the scalability of this ultra-fast format to about 20 bits or less, for current memory technology. Type II unums are also are much less amenable to *fused* operations. These drawbacks motivated a search for a format that would keep many of the merits of Type II unums, but be "hardware friendly," that is, computable using existing float-like logic.

2. Posits and Valids

We contrast two esthetics for calculation involving real numbers:

- Non-rigorous, but cheap, fast and "good enough" for an established set of applications
- Rigorous and mathematical, even at the cost of greater execution time and storage

The first esthetic has long been addressed by float arithmetic, where rounding error is tolerated, and the second esthetic has been addressed by interval arithmetic. Type I and Type II unums can do either, which is one reason they are "universal numbers." However, if we are always going to use the "guess" function to round after every operation, we are better off using the last bit as another significant fraction bit and not as the ubit. A unum of this type is what we call a *posit*. To quote the New Oxford American Dictionary (Third Edition):

posit (noun): a statement that is made on the assumption that it will prove to be true.

A hardware-friendly version of Type II unums relaxes one of the rules: Reciprocals only follow the perfect reflection rule for $0, \pm\infty$, and integer powers of 2. This frees us to populate the u-lattice in a way that keeps the finite numbers *float-like*, in that they are all of the form $m \cdot 2^k$ where k and m are integers. There are no open intervals.

A *valid* is a pair of equal-size posits, each ending in a ubit. They are intended for use where applications need the rigor of interval-type bounds, such as when debugging a numerical algorithm. Valids are more powerful than traditional interval arithmetic and less prone to rapidly expanding, overly pessimistic bounds [2, 4]. They are not the focus of this paper, however.

2.1. The Posit Format

Here is the structure of an n-bit posit representation with es exponent bits (fig. 2).

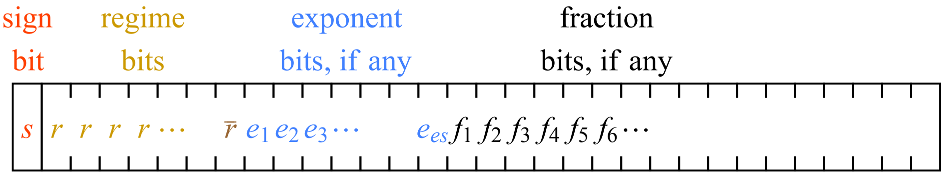

Figure 2. Generic posit format for finite, nonzero values

The sign bit is what we are used to: **0** for positive numbers, **1** for negative numbers. If negative, take the 2's complement before decoding the regime, exponent, and fraction.

To understand the *regime* bits, consider the binary strings shown in Table 1, with numerical meaning k determined by the *run length* of the bits. (An "x" in a bit string means, "don't care").

Table 1. Run-length meaning k of the regime bits

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	−4	−3	−2	−1	0	1	2	3

We call these leading bits the *regime* of the number. Binary strings begin with some number of all **0** or all **1** bits in a row, terminated either when the next bit is opposite, or the end of the string is reached. Regime bits are color-coded in **amber** for the identical bits r, and **brown** for the opposite bit $\bar{r}$ that terminates the run, if any. Let m be the number of identical bits in the run; if the bits are **0**, then $k = -m$; if they are **1**, then $k = m - 1$. Most processors can "find first **1**" or "find first **0**" in hardware, so decoding logic for regime bits is readily available. The regime indicates a scale factor of $useed^k$, where $useed = 2^{2^{es}}$. Table 2 shows example $useed$ values.

Table 2. Table 1. The $useed$ as a function of es

es	0	1	2	3	4
$useed$	2	$2^2 = 4$	$4^2 = 16$	$16^2 = 256$	$256^2 = 65536$

The next bits (color-coded **blue**) are the exponent e, regarded as an unsigned integer. There is no bias as there is for floats; they represent scaling by 2^e. There can be up to es exponent bits, depending on how many bits remain to the right of the regime. This is a compact way of expressing *tapered accuracy*; numbers near 1 in magnitude have more accuracy than extremely large or extremely small numbers, which are much less common in calculations.

If there are any bits remaining after the regime and the exponent bits, they represent the fraction, f, just like the fraction **1**.f in a float, with a hidden bit that is always **1**. There are no subnormal numbers with a hidden bit of **0** as there are with floats.

The system just described is a natural consequence of populating the u-lattice. Start from a simple 3-bit posit; for clarity, fig. 3 shows only the right half of the projective reals. So far, fig. 3 follows Type II rules. There are only two posit *exception values*: 0 (all **0** bits) and $\pm\infty$ (**1** followed by all **0** bits), and their bit string meanings do not follow positional notation. For the other posits in fig. 3, the bits are color-coded as described above. Note that positive values in fig. 3 are exactly $useed$ to the power of the k value represented by the regime.

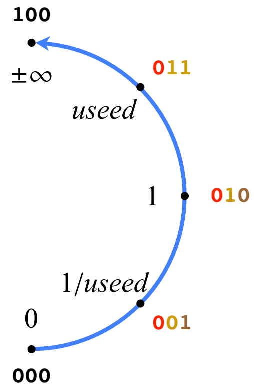

Figure 3. Positive values for a 3-bit posit

Posit precision increases by appending bits, and values remain where they are on the circle when a **0** bit is appended. Appending a **1** bit creates a new value between two posits on the circle. What value should we assign to each in-between value? Let *maxpos* be the largest positive value and *minpos* be the smallest positive value on the ring defined with a bit string. In fig. 3, *maxpos* is *useed* and *minpos* is 1/*useed*. The interpolation rules are as follows:

- Between the *maxpos* and ±∞, the new value is *maxpos* × *useed*; and between 0 and *minpos*, the new value is *minpos*/*useed* (new **regime** bit).
- Between existing values $x = 2^m$ and $y = 2^n$ where m and n differ by more than 1, the new value is their *geometric* mean, $\sqrt{x \cdot y} = 2^{(m+n)/2}$ (new **exponent** bit).
- Otherwise, the new value is midway between the existing x and y values next to it, that is, it represents the *arithmetic* mean $(x + y)/2$ (new **fraction** bit).

As an example, fig. 4 shows a build up from a 3-bit to a 5-bit posit with $es = 2$, so $useed = 16$:

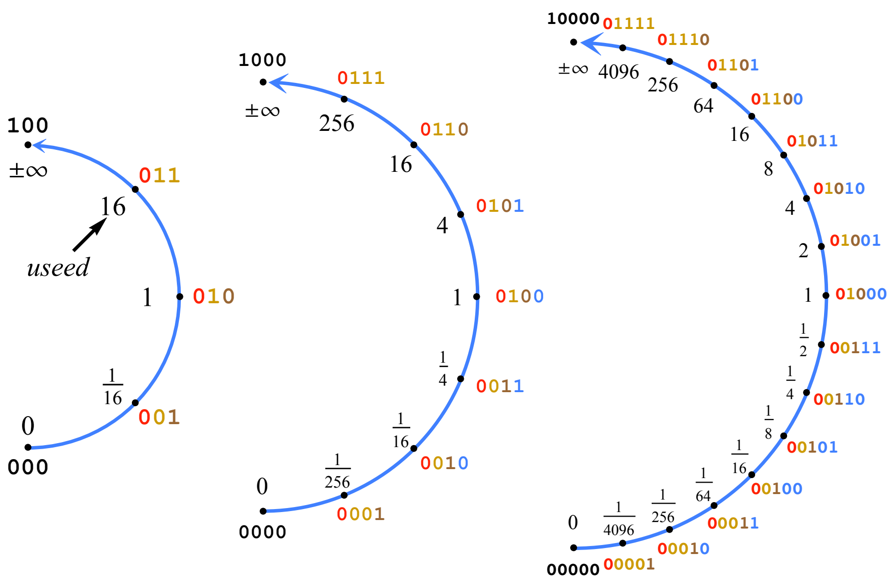

Figure 4. Posit construction with two exponent bits, $es = 2$, $useed = 2^{2^{es}} = 16$

If one more bit were appended in fig. 4 to make 6-bit posits, posits representing the range of values between 1/16 and 16 will append **fraction** bits, not exponent bits.

Suppose we view the bit string for a posit p as a signed integer, ranging from -2^{n-1} to $2^{n-1}-1$. Let k be the integer represented by the regime bits, e be the unsigned integer represented by the exponent bits, if any. If the set of fraction bits is $\{f_1 f_2 \ldots f_{fs}\}$, possibly the empty set, let f be the value represented by $\mathtt{1.}f_1 f_2 \ldots f_{fs}$. Then p represents

$$x = \begin{cases} 0, & p = 0, \\ \pm\infty, & p = -2^{n-1}, \\ sign(p) \times useed^k \times 2^e \times f, & \text{all other } p. \end{cases}$$

The regime and *es* bits serve the function of the exponent bits in a standard float; together, they set the power-of-2 scaling of the fraction where each *useed* increment is a batch shift of 2^{es} bits. The *maxpos* is $useed^{n-2}$ and the *minpos* is $useed^{2-n}$. An example decoding of a posit is shown in fig. 5 (with a "nonstandard" value for *es* here, for clarity).

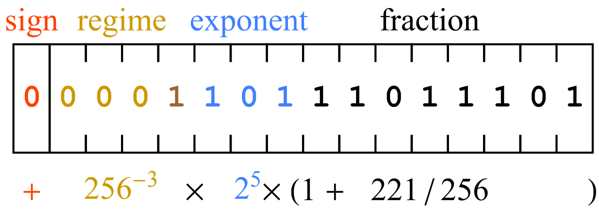

Figure 5. Example of a posit bit string and its mathematical meaning

The sign bit **0** means the value is positive. The regime bits **0001** have a run of three **0**s, which means k is -3; hence, the scale factor contributed by the regime is 256^{-3}. The exponent bits, **101**, represent 5 as an unsigned binary integer, and contribute another scale factor of 2^5. Lastly, the fraction bits **11011101** represent 221 as an unsigned binary integer, so the fraction is $1 + 221/256$. The expression shown underneath the bit fields in fig. 5 works out to $477/134217728 \approx 3.55393 \times 10^{-6}$.

2.2. 8-bit Posits and Neural Network Training

While IEEE floats do not define a "quarter-precision" 8-bit float, an 8-bit posit with $es = 0$ has proved to be surprisingly useful for some purposes; they are sufficiently powerful to *train neural networks* [3, 8]. Currently, half-precision (16-bit) IEEE floats are often used for this purpose, but 8-bit posits have the potential to be $2-4\times$ faster. An important function for neural network training is a *sigmoid function*, a function $f(x)$ that is asymptotically 0 as $x \to -\infty$ and asymptotically 1 as $x \to \infty$. A common sigmoid function is $1/(1 + e^{-x})$ which is expensive to compute, easily requiring over a hundred clock cycles because of the math library call to evaluate $\exp(x)$, and because of the divide. With posits, you can simply flip the first bit of the posit representing x, shift it two bits to the right (shifting in **0** bits on the left), and the resulting posit function in fig. 6 (shown in **magenta**) closely resembles $1/(1 + e^{-x})$ (shown in **green**); it even has the correct slope where it intersects the y-axis.

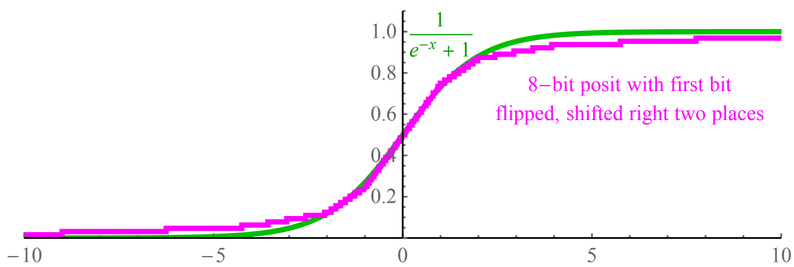

Figure 6. Fast sigmoid function using posit representation

2.3. Using the Useed to Match or Exceed the Dynamic Range of Floats

We define the *dynamic range* of a number system as the number of decades from the smallest to largest positive finite values, *minpos* to *maxpos*. That is, the dynamic range is defined as $\log_{10}(maxpos) - \log_{10}(minpos) = \log_{10}(maxpos/minpos)$. For an 8-bit posit system with $es = 0$, *minpos* is 1/64 and *maxpos* is 64, so the dynamic range is about 3.6 decades. Posits defined with $es = 0$ are elegant and simple, but their 16-bit and larger versions have less dynamic range than an IEEE float of the same size. For example, a 32-bit IEEE float has a dynamic range of about 83 decades, but a 32-bit posit with $es = 0$ will have only about 18 decades of dynamic range.

Here is a table of *es* values that allow posits to surpass the dynamic range of floats for 16-bit and 32-bit size, and closely match it for 64-bit, 128-bit, and 256-bit sizes.

Table 3. Float and posit dynamic ranges for the same number of bits

Size, Bits	**IEEE Float Exp. Size**	**Approx. IEEE Float Dynamic Range**	**Posit *es* Value**	**Approx. Posit Dynamic Range**
16	5	6×10^{-8} to 7×10^{4}	1	4×10^{-9} to 3×10^{8}
32	8	1×10^{-45} to 3×10^{38}	3	6×10^{-73} to 2×10^{72}
64	11	5×10^{-324} to 2×10^{308}	4	2×10^{-299} to 4×10^{298}
128	15	6×10^{-4966} to 1×10^{4932}	7	1×10^{-4855} to 1×10^{4855}
256	19	2×10^{-78984} to 2×10^{78913}	10	2×10^{-78297} to 5×10^{78296}

One reason for choosing $es = 3$ for 32-bit posits is to make it easier to substitute them not just for 32-bit floats, but for 64-bit floats as well. Similarly, the 17-decade dynamic range of 16-bit posits opens them up to applications currently are tackled only with 32-bit floats. We will show that posits can exceed both the dynamic range *and* accuracy of floats with the same bit width.

2.4. Qualitative Comparison of Float and Posit Formats

There are no "NaN" (not-a-number) bit representations with posits; instead, the calculation is interrupted, and the interrupt handler can be set to report the error and its cause, or invoke a workaround and continue computing, but posits do *not* make the logical error of assigning a number to something that is, by definition, not a number. This simplifies the hardware considerably. If a programmer finds the need for NaN values, it indicates the program is not yet finished, and the use of *valids* should be invoked as a sort of numerical debugging environment to find and eliminate possible sources of such outputs. Similarly, posits lack a separate ∞ and $-\infty$ like floats have; however, *valids* support open intervals $(maxpos, \infty)$ and $(-\infty, -maxpos)$, which provide the ability to express unbounded results of either sign, so the need for signed infinity is once again an indication that valids are called for instead of posits.

There is no "negative zero" in posit representation; "negative zero" is another defiance of mathematical logic that exists in IEEE floats. With posits, when $a = b$, $f(a) = f(b)$. The IEEE 754 standard says that the reciprocal of "negative zero" is $-\infty$ but the reciprocal of "positive zero" is ∞, but also says that negative zero equals positive zero. Hence, floats imply that $-\infty = \infty$.

Floats have a complicated test for equality, $a \stackrel{?}{=} b$. If either a or b are NaN, the result is always false *even if the bit patterns are identical.* If the bit patterns are *different*, it is still possible for a to equal b, since negative zero equals positive zero! With posits, the equality test is exactly the same as comparing two integers: if the bits are the same, they are equal. If any bits differ, they are not equal. Posits share the same $a < b$ relation as signed integers; as with signed integers, you have to watch out for wraparound, but you really don't need separate machine instructions for posit comparisons if you already have them for signed integers.

There are no *subnormal* numbers in the posit format, that is, special bit patterns indicating that the hidden bit is `0` instead of `1`. Posits do not use "gradual underflow." Instead, they used tapered precision, which provides the functionality of gradual underflow and a symmetrical counterpart, gradual overflow. (Instead of gradual overflow, floats are asymmetric and use those bit patterns for a vast and unused cornucopia of NaN values.)

Floats have one advantage over posits for the hardware designer: the fixed location of bits for the exponent and the fraction mean they can be decoded in parallel. With posits, there is a little serialization in having to determine the regime bits before the other bits can be decoded. There is a simple workaround for this in a processor design, similar to a trick used to speed the exception handling of floats: Some extra register bits can be attached to each value to save the need for extracting size information when decoding instructions.

3. Bitwise Compatibility and Fused Operations

One reason IEEE floats do not give identical results across systems is because elementary functions like $\log(x)$ and $\cos(x)$ are not required by IEEE to be accurate to the last bit for every possible input. Posit environments must correctly round *all* supported arithmetic operations. (Some math library programmers worry about "The Table-Maker's Dilemma" that some values can take much more work to determine their correct rounding; this can be eliminated by using interpolation tables instead of polynomial approximations.) An incorrect value in the last bit for e^x, say, should be treated the way we would treat a computer system that tells us $2 + 2 = 5$.

The more fundamental reason IEEE floats fail to give repeatable results from system to system is because the standard permits *covert* methods to avoid overflow/underflow and perform operations more accurately, such as by internally carrying extra bits in the exponent or fraction. Posit arithmetic forbids such covert help.

The most recent version (2008) of the IEEE 754 standard [7] includes the fused multiply-add in its requirements. This was a controversial change, and vehemently opposed by many of the committee members. *Fusing* means deferring rounding until the last operation in a computation involving more than one operation, after performing all operations using exact integer arithmetic in a scratch area with a set size. Fusing is not the same as general extended-precision arithmetic, which can increase the size of integers until the computer runs out of memory.

The posit environment **mandates** the following fused operations:

Fused multiply-add	$(a \times b) + c$
Fused add-multiply	$(a + b) \times c$
Fused multiply-multiply-subtract	$(a \times b) - (c \times d)$
Fused sum	$\sum a_i$
Fused dot product (scalar product)	$\sum a_i b_i$

Note that all of the operations in the above list are subsets of the fused dot product [6] in terms of processor hardware requirements. The smallest magnitude nonzero value that can arise in doing a dot product is $minpos^2$. Every product is an integer multiple of $minpos^2$. If we have to perform the dot product of vectors $\{maxpos, minpos\}$ and $\{maxpos, minpos\}$ as an exact operation in a scratch area, we need an integer big enough to hold $maxpos^2/minpos^2$. Recall that

$maxpos = useed^{n-2}$ and $minpos = 1/maxpos$. Hence, $maxpos^2/minpos^2 = useed^{4n-8}$. Allowing for carry bits, and rounding up to a power of 2, tab. 4 shows recommended accumulator sizes.

Table 4. Exact accumulator sizes for each posit size

Posit size in bits, n	16	32	64	128	256
Exact accumulator size, bits	128	1024	4096	65536	1048576

Some accumulators clearly are register-sized data, while the larger accumulators will require a scratch area equivalent of L1 or L2 cache. The fused operations can be done either through software or hardware, but must be available for a posit environment to be complete.

4. Quantitative Comparisons of Number Systems

4.1. A Definition of *Decimal Accuracy*

Accuracy is the inverse of error. If we have a pair of numbers x and y (nonzero and of the same sign), their order-of-magnitude distance is $|\log_{10}(x/y)|$ decades, the same measure that defines the dynamic range with smallest and largest representable positive numbers x and y. A "perfect spacing" of ten numbers between 1 and 10 in a real number system would be not the evenly-spaced counting numbers 1 through 10, but exponentially-spaced $1, 10^{1/10}, 10^{2/10}, \ldots, 10^{9/10}, 10$. That is the *decibel scale*, long used by engineers to define ratios; for example, ten decibels is a factor of 10. Thirty decibels (30 dB) means a factor of $10^3 = 1000$. The ratio 1 dB is about $1.26\cdots$; if you know a value to within 1 dB, then you have 1 decimal of accuracy. If you knew it to 0.1 dB, that would mean 2 decimals of accuracy, and so on. The formula for *decimal accuracy* is $\log_{10}(1/|\log_{10}(x/y)|) = -\log_{10}(|\log_{10}(x/y)|)$, where x and y are either the correct value and the computed value when using a rounding system like floats and posits, or the lower and upper limits of a bound, if using a rigorous system like intervals or valids.

4.2. Defining Float and Posit Comparison Sets

We can create "scale models" of both floats and posits with only 8 bits each. The advantage of this approach is that the 256 values produced are a small enough set that we can *exhaustively* test and compare all 256^2 entries in the tables for $+ \; - \; \times \; \div$ for numerical properties. A "quarter-precision" float with a sign bit, four-bit exponent field and three-bit fraction field can adhere to all the rules of the IEEE 754 standard. Its smallest positive real (a subnormal float) is 1/1024 and its largest positive real is 240, an asymmetrical dynamic range of about 5.1 decades. Fourteen of the bit patterns represent NaN.

The comparable 8-bit posit environment uses $es = 1$; its positive reals range from 1/4096 to 4096, a symmetrical dynamic range of about 7.2 decades. There are no NaN values. We can plot the decimal accuracy of the positive numbers in both sets, as shown in fig. 7. Note that the values represented by posits represent over two decades more dynamic range than the floats, as well as being as accurate or more for all but the values where floats are close to underflow or overflow. The jaggedness is characteristic of all number systems that approximate a logarithmic representation with piecewise linear sequences. Floats have tapered accuracy only on the left, using gradual underflow; on the right, accuracy falls off a cliff to accommodate all the NaN values. Posits come much closer to a symmetrical tapered accuracy.

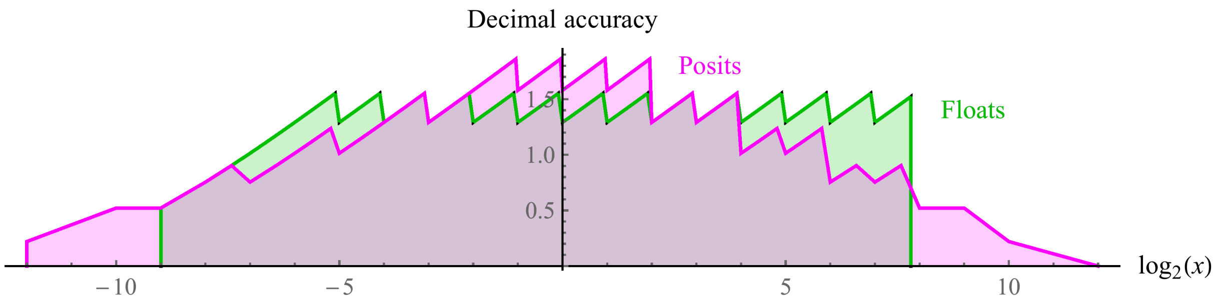

Figure 7. Decimal accuracy comparison of floats and posits

4.3. Single-Argument Operation Comparisons

4.3.1. Reciprocal

For every possible input value x to the function $1/x$, the result can land exactly on another number in the set or it can round, in which case we can measure the decimal loss using the formula of Section 4.1; for floats, it can overflow or produce a NaN. See fig. 8.

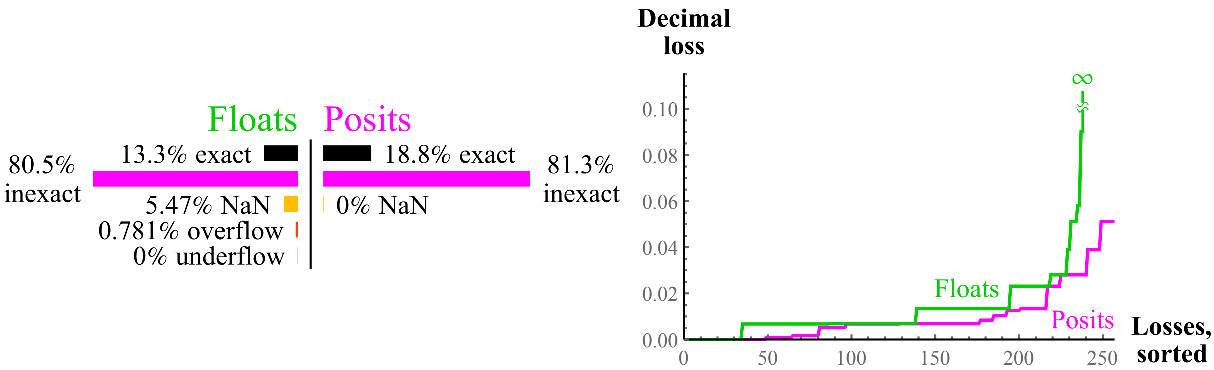

Figure 8. Quantitative comparison of floats and posits computing the reciprocal, $1/x$

The graph on the right plots every decimal accuracy loss created by reciprocation, other than the float cases that produce NaN. Posits are superior to floats, with far more exact cases, and they retain that superiority through their entire range of sorted losses. Subnormal floats overflow under reciprocation, which creates an infinite loss of decimal accuracy, and of course the float NaN *input* values produce NaN output values. Posits are *closed under reciprocation.*

4.3.2. Square Root

The square root function does not underflow or overflow. For negative values, and for the NaN inputs for floats, the result is NaN. Remember that these are "scale model" 8-bit floats and posits; the advantage of posits **increases** with data precision. For a similar plot of 64-bit floats vs. posits, posit error would be about 1/30 of float error instead of about ½ the float error.

4.3.3. Square

Another common unary operation is x^2. Overflow and underflow are a common disaster when squaring floats. For almost half the float cases, squaring does not produce a meaningful numerical value, whereas every posit can be squared to produce another posit. (The square of unsigned infinity is again unsigned infinity.)

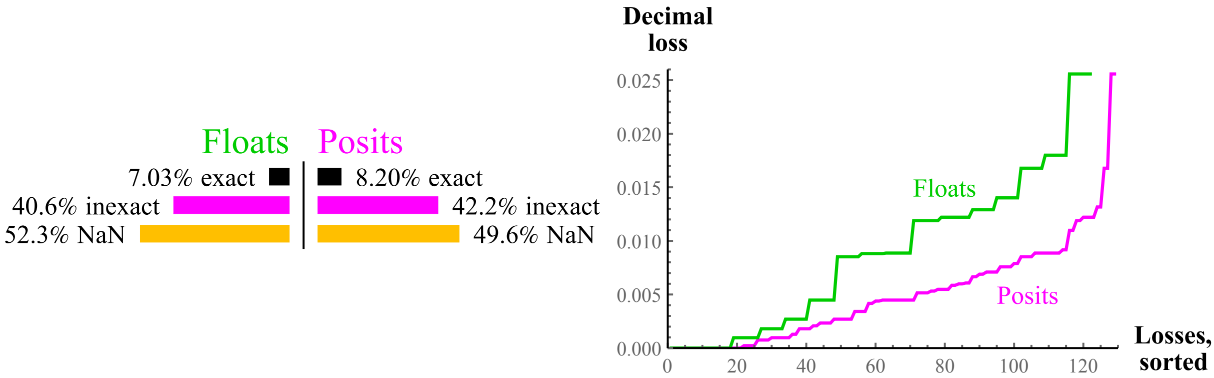

Figure 9. Quantitative comparison of floats and posits computing $\sqrt{x}$

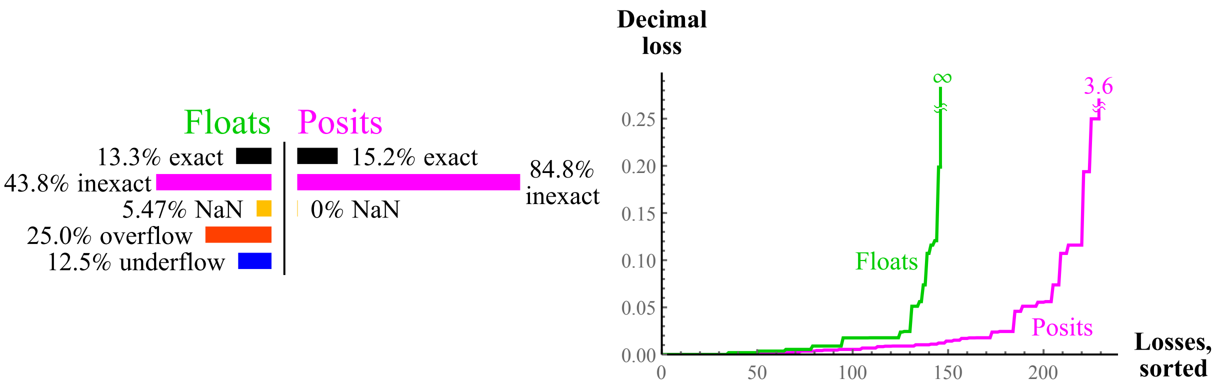

Figure 10. Quantitative comparison of floats and posits computing x^2

4.3.4. Logarithm Base 2

We can also compare logarithm base 2 closure, that is, the percentage of cases where $\log_2(x)$ is exactly representable and how much decimal accuracy is lost when it is not exact. Floats actually have one advantage in that they can represent $\log_2(0)$ as $-\infty$ and $\log_2(\infty)$ as ∞, but this is more than compensated by the richer vocabulary of integer powers of 2 for posits.

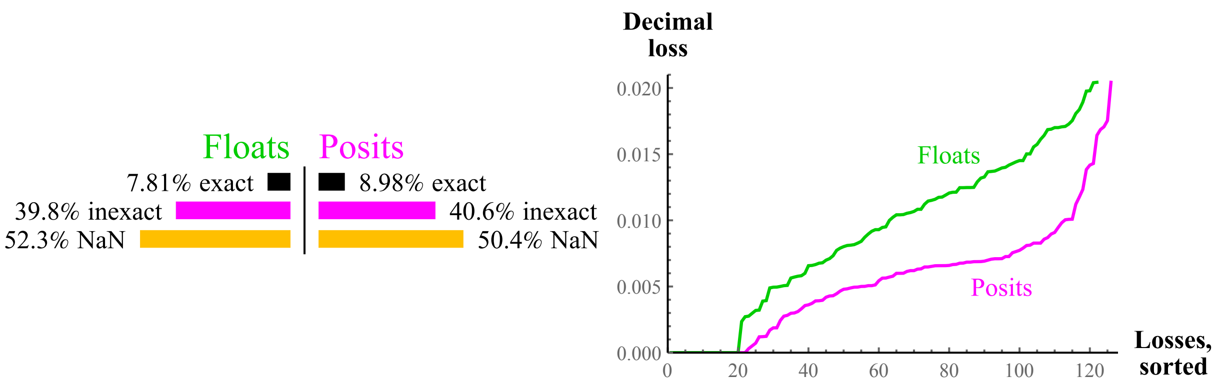

Figure 11. Quantitative comparison of floats and posits computing $\log_2(x)$

The graph is similar to that for square root, with about half the input values resulting in NaN in both cases, but posits experience about half the loss of decimal accuracy. If you can compute $\log_2(x)$, it just takes a scale factor to convert to $\ln(x)$ or $\log_{10}(x)$ or any other logarithm base.

4.3.5. *Exponential,* 2^x

Similarly, once you can compute 2^x, it is easy to derive a scale factor that also gets you e^x or 10^x and so on. Posits have just one exception case: 2^x is NaN when the argument is $\pm\infty$.

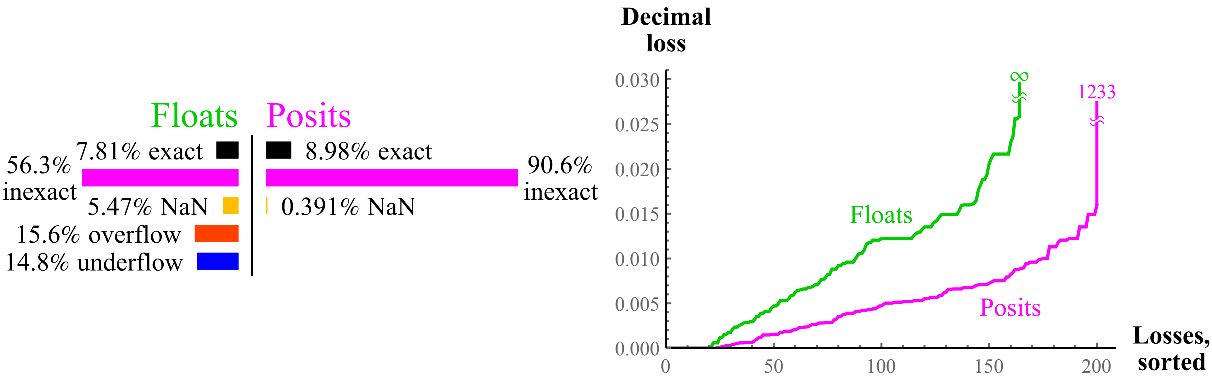

Figure 12. Quantitative comparison of floats and posits computing 2^x

The maximum decimal loss for posits seems large, because 2^{maxpos} will be rounded back to *maxpos*. For this example set, just a few errors are as high as $\log_{10}(2^{4096}) \approx 1233$ decimals. Consider which is worse: the loss of over a thousand decimals, or the loss of an *infinite* number of decimals? If you can stay away from those very largest values, posits are still a win, because the error for smaller values is much better behaved. The only time you get a large decimal loss with posits is when working with numbers *far outside of what floats can even express* as input arguments. The graph shows how posits are more robust in terms of the dynamic range for which results are reasonable, and the superior decimal accuracy throughout this range.

For common unary operations $1/x, \sqrt{x}, x^2, \log_2(x)$, and 2^x, posits are consistently and uniformly more accurate than floats with the same number of bits, and produce meaningful results over a larger dynamic range. We now turn our attention to the four elementary arithmetic operations that take two arguments: Addition, subtraction, multiplication, and division.

4.4. Two-Argument Operation Comparisons

We can use the scale-model number systems to examine two-argument arithmetic operations like $+\ -\ \times\ \div$. To help visualize all 65536 results, we make 256 by 256 "closure plots" that show at a glance what fraction of the results are exact, inexact, overflow, underflow, or NaN.

4.4.1. *Addition and Subtraction*

Because $x - y = x + (-y)$ works perfectly in both floats and posits, there is no need to study subtraction separately. For the addition operation, we compute $z = x + y$ exactly, and compare it to the sum that is returned by the rules of each number system. It can happen that the result is exact, that it must be rounded to a nearby finite nonzero number, that it can overflow or underflow, or can be an indeterminate form like $\infty - \infty$ that produces a NaN. Each of these is color-coded so we can look at the entire addition table at a glance. In the case of rounded results, the color-coding is a gradient from black (exact) to magenta (maximum error of either posits or floats). fig. 13 shows what the closure plots look like for the floats and the unums.

As with the unary operations, but with far more data points, we can summarize the ability of each number system to produce meaningful and accurate answers:

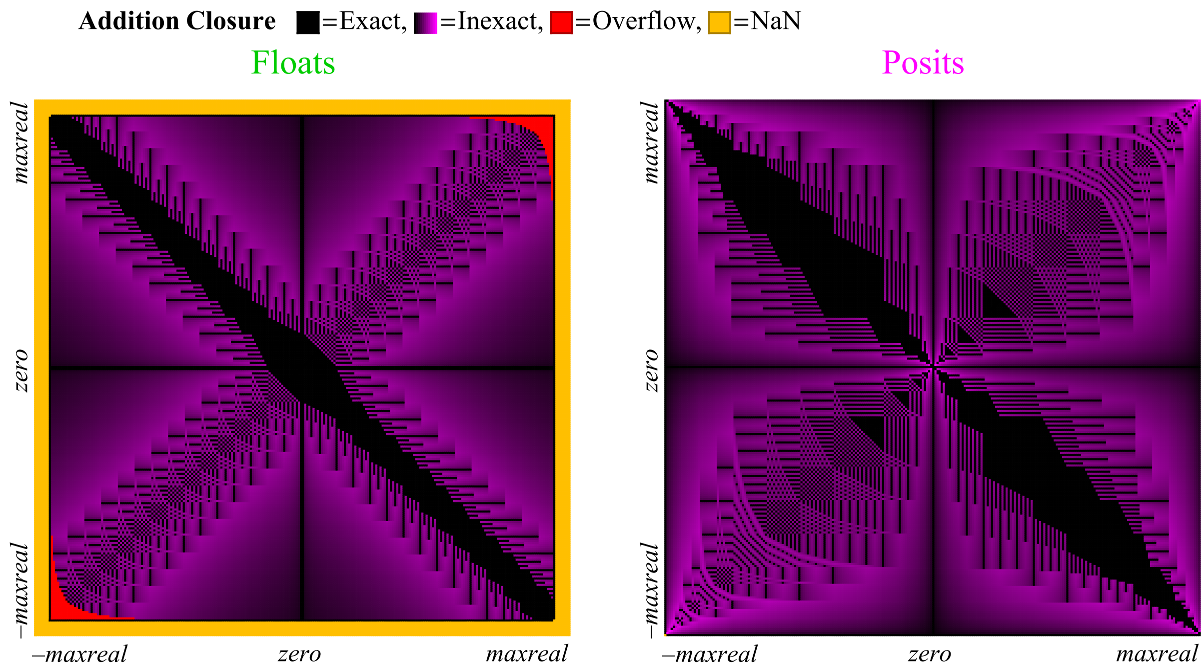

Figure 13. Complete closure plots for float and posit addition tables

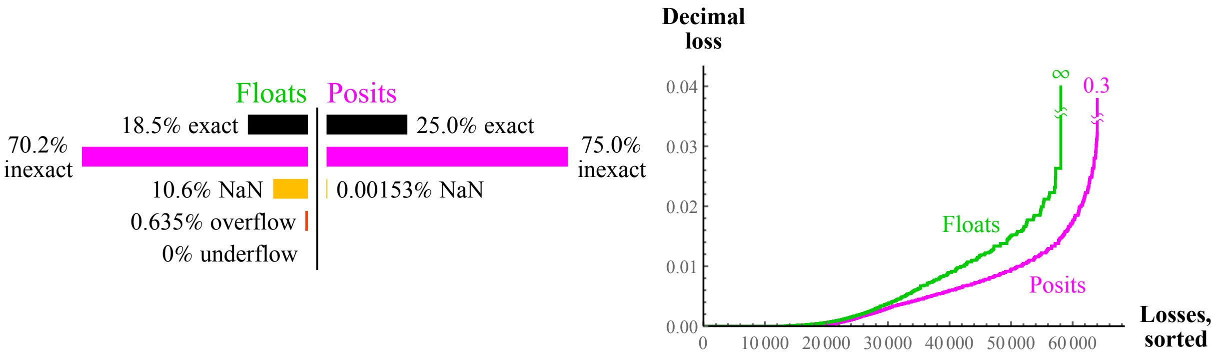

Figure 14. Quantitative comparison of floats and posits for addition

It is obvious at a glance that posits have significantly more additions that are exact. The broad black diagonal band in the float closure plot is much wider than it would be with higher precision, because it represent the gradual underflow region where floats are equally spaced like fixed-point numbers; that is a large fraction of all possible floats when we only have 8 bits.

4.4.2. Multiplication

We use a similar approach to compare how well floats and posits multiply. Unlike addition, multiplication can cause floats to *underflow*. The "gradual underflow" region provides some protection, as you can see in the center of the float closure graph (fig. 15, left). Without it, the blue underflow region would be a full diamond shape. The posit multiplication closure plot is much less colorful, which is a good thing. Only two pixels light up as NaN, near where the axes have their "zero" label. That is where $\pm\infty \times 0$ occurs. Floats have more cases where the product is exact than do posits, *but at a terrible cost.* As fig. 15 shows, almost one-quarter of all float products overflow or underflow, and that fraction does not decrease for higher precision floats.

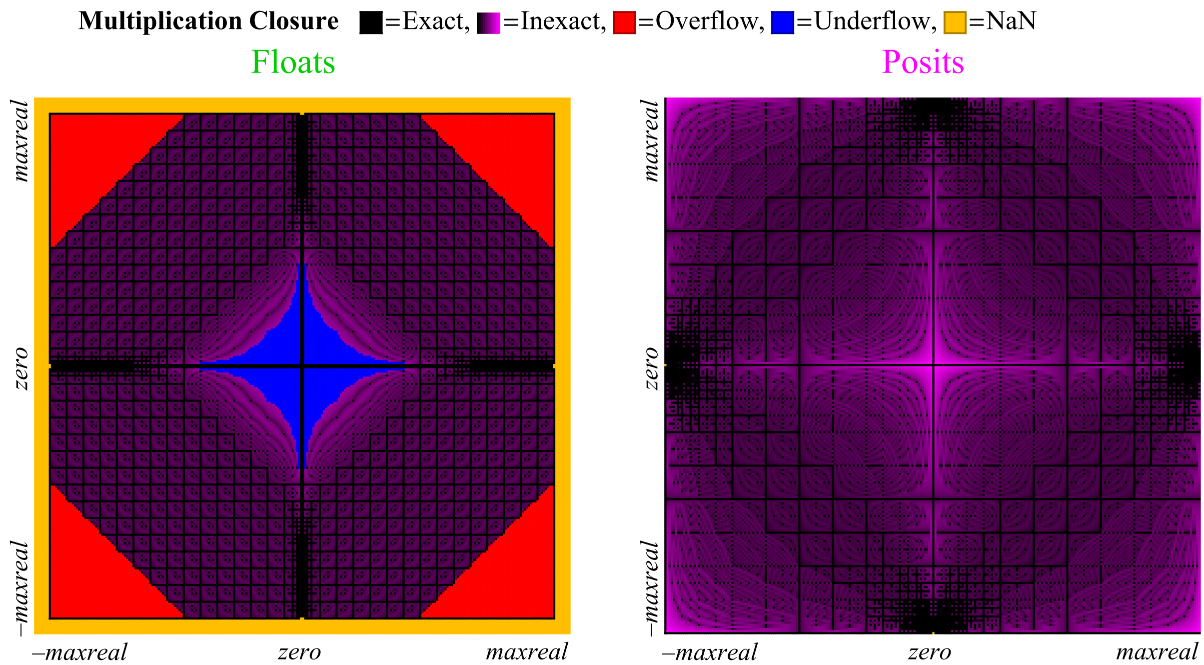

Figure 15. Complete closure plots for float and posit multiplication tables

The worst-case rounding for posits occurs for *maxpos* × *maxpos*, which is rounded back to *maxpos*. For these posits, that represents a (very rare) loss of about 3.6 decimals. As the graph in fig. 16 shows, posits are dramatically better at minimizing multiplication error than floats.

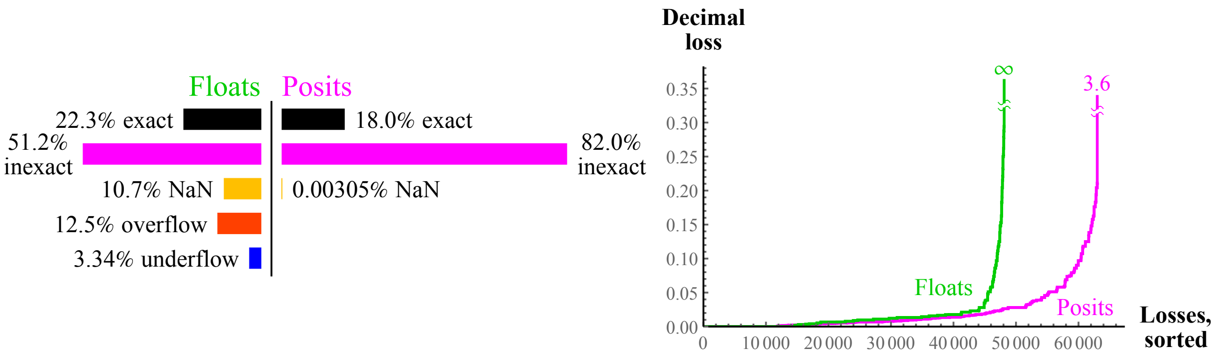

Figure 16. Quantitative comparison of floats and posits for multiplication

The closure plots for the division operation are like those for multiplication, but with the regions permuted; to save space, they will not be shown here. The quantitative comparison for division is almost identical to that for multiplication.

4.5. Comparing Floats and Posits in Evaluating Expressions

4.5.1. The "Accuracy on a 32-Bit Budget" Benchmark

Benchmarks usually make a goal of the smallest execution time, and are frequently vague about how accurate the answer has to be. A different kind of benchmark is one where we fix the precision budget, that is, the number of bits per variable, and try to get the maximum decimal accuracy in the result. Here is an example of an expression we can use to compare various number systems with a 32-bit budget per number:

$$X = \left(\frac{27/10 - e}{\pi - (\sqrt{2} + \sqrt{3})} \right)^{67/16} = 302.8827196\cdots$$

The rule is that we start with the best representation the number system has for e and π, and for all the integers shown, and see how many decimals agree with the correct value for X after performing the nine operations in the expression. We will show incorrect digits in orange.

While IEEE 32-bit floats have decimal accuracy that wobbles between 7.3 and 7.6 decimals, the accumulation of rounding error in evaluating X gives the answer 302.912⋯, with only three correct digits. This is one reason IEEE float users feel pressure to use 64-bit floats everywhere, because even simple expressions risk losing so much accuracy that the results might be useless.

The 32-bit posits have tapered decimal accuracy, with a wobble between about 8.2 and 8.5 decimals for numbers near 1 in magnitude. In evaluating X, they give the answer 302.88231⋯, twice as many significant digits. Bear in mind that 32-bit posits have a dynamic range of over 144 decades, whereas 32-bit IEEE floats have a much smaller dynamic range of about 83 decades. Therefore, the extra accuracy in the result was *not* attained at the expense of dynamic range.

4.5.2. A Quad-Precision Test: Goldberg's Thin Triangle Problem

Here's a classic "thin triangle" problem [1]: Find the area of a triangle with sides a, b, c, when two of the sides b and c are just 3 Units in the Last Place (ULPs) longer than half the longest side a (fig. 17):

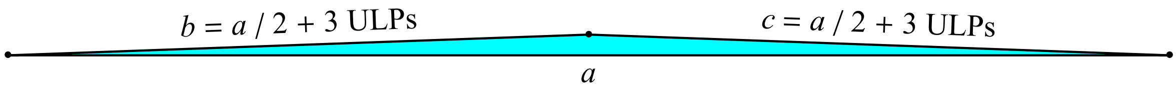

Figure 17. Goldberg's thin triangle problem

The classic formula for the area A uses a temporary value s:

$$s = \frac{a+b+c}{2}; A = \sqrt{s(s-a)(s-b)(s-c)}$$

The hazard in the formula for a thin triangle is that s is very close to the value of a, so the calculation of $(s - a)$ magnifies any rounding into a large relative error. Let's try 128-bit (quad-precision) IEEE floats, where $a = 7, b = c = 7/2 + 3 \times 2^{-111}$. (If the units are in light-years, then the short sides are only longer than half the long side by 1/200 the diameter of a proton. Yet that pops the triangle up to about the width of a doorway at the thickest point.) We also evaluate the formula for A using 128-bit posits ($es = 7$). Here are the results:

Correct answer:	$3.14784204874900425235885265494550774498\cdots \times 10^{-16}$
128-bit IEEE float answer:	$3.63481490842332134725920516158057682788\cdots \times 10^{-16}$
128-bit posit answer:	$3.14784204874900425235885265494550774439\cdots \times 10^{-16}$

Posits have as much as 1.8 decimal digits more accuracy than floats in quad precision over a vast dynamic range: from 2×10^{-270} to 5×10^{-269}. That is ample for protecting against this particular inaccuracy-amplifying disaster. It is interesting to note that the posit answer would be more accurate than the float answer even if it were converted to a 16-bit posit at the end.

4.5.3. *The Quadratic Formula*

There is a classic numerical analysis "trick" to avoid a rounding error hazard in finding the roots r_1, r_2 of $ax^2 + bx + c = 0$ using the usual formula $r_1, r_2 = (-b \pm \sqrt{b^2 - 4ac})/(2a)$ when b is much larger than a and c, resulting in left-digit cancellation for the root for which $\sqrt{b^2 - 4ac}$ is very close to b. Instead of expecting programmers to memorize a panoply of arcane tricks for every formula, perhaps posits can make it a bit safer to simply apply the textbook form of the formula. Suppose $a = 3, b = 100, c = 2$, and compare 32-bit IEEE floats with 32-bit posits.

Table 5. Quadratic equation evaluation

Root	**Mathematical Value**	**Float result**	**Posit result**
r_1	$(-100 + \sqrt{9976})/6 = -0.0200120144\cdots$	-0.02001190	$-0.02001206\cdots$
r_2	$(-100 - \sqrt{9976})/6 = -33.3133213\cdots$	-33.313302	$-33.3133216\cdots$

The numerically unstable root is r_1, but notice that 32-bit posits still return 6 correct decimals instead of only 4 correct decimals from the float calculation.

4.6. Comparing Floats and Posits for the Classic LINPACK Benchmark

The basis for ranking supercomputers has long been the time to solve an n-by-n system of linear equations $\mathbf{A}x = b$. Specifically, the benchmark populates a matrix $\mathbf{A}$ with pseudo-random numbers between 0 and 1, and sets the b vector to the row sums of $\mathbf{A}$. That means the solution x should be a vector of all 1s. The benchmark driver calculates the norm of the residual $||\mathbf{A}x - b||$ to check correctness, though there is no "hard number" in the rules that limits how inaccurate the answer can be. It is typical to lose several decimals of accuracy in the calculation, which is why it requires 64-bit floats to run (not necessarily IEEE). The original benchmark size was $n = 100$, but that size became too small for the fastest supercomputers so n was increased to 300, and then to 1000, and finally (at the urging of the first author) changed to a *scalable* benchmark that ranks based on operations per second assuming $^2/_3 n^3 + 2n^2$ multiply or add operations.

In comparing floats and posits, we noticed a subtle flaw in the benchmark driver: The answer is generally *not* a sequence of all 1 values, because *rounding occurs when computing the row sums.* That error can be eliminated by finding which entries in $\mathbf{A}$ contribute a `1` bit to the sum beyond the precision capabilities, and setting that bit to `0`. This assures that the row sum of $\mathbf{A}$ is representable without rounding, and then the answer x actually **is** a vector of all 1s.

For the original 100-by-100 problem, 64-bit IEEE floats produce an answer that looks like

$$\begin{array}{l} 0.9999999999999633626401873698341660201549530029296875 \\ 1.0000000000000111022302462515654042363166809082031250 \\ \vdots \\ 1.0000000000000226485497023531934246420860290527343750 \end{array}$$

Not a single one of the 100 entries is correct; they are close to 1, but never land on it. With posits, we can do a remarkable thing. Use *32-bit* posits and the same algorithm; calculate the residual, $r = \mathbf{A}x - b$, using the fused dot product. Then solve $\mathbf{A}x' = r$ (using the already-factored $\mathbf{A}$) and use x'as a correction: $x \leftarrow x - x'$. The result is an unprecedented **exact** answer to the LINPACK benchmark: $\{1, 1, \ldots, 1\}$. Can the rules of LINPACK forbid the use of a new 32-bit number type that can get the perfect result with a residual of *zero*, and continue to insist on the use of 64-bit float representations that cannot? That decision will be up to the shepherds of the benchmark. For those who use linear equations to solve real problems and not to compare supercomputer speeds, posits offer an overwhelming advantage.

5. Summary

Posits beat floats at their own game: guessing their way through a calculation while incurring rounding errors. Posits have higher accuracy, larger dynamic range, and better closure. They can be used to produce better answers with the same number of bits as floats, or (what may be even more compelling) an equally good answer with *fewer bits*. Since current systems are bandwidth-limited, using smaller operands means higher speed and less power use.

Because they work like floats, not intervals, they can be regarded as a drop-in replacement for floats, as demonstrated here. If an algorithm has survived the test of time as stable and "good enough" using floats, then it will run even better with posits. The fused operations available in posits provide a powerful way to prevent rounding error from accumulating, and in some cases may allow us to safely use 32-bit posits instead of 64-bit floats in high-performance computing. Doing so will generally increase the speed of a calculation $2 - 4\times$, and save energy and power as well as the cost of storage. Hardware support for posits will give us the equivalent of one or two turns of Moore's law improvement, without any need to reduce transistor size and cost.

Unlike floats, posits produce bitwise-reproducible answers across computing systems, overcoming the primary failure of the IEEE 754 float definition. The simpler and more elegant design of posits compared to floats reduces the circuitry needed for fast hardware. As ubiquitous as floats are today, posits could soon prove them obsolete. □

References

1. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991. DOI: doi:10.1145/103162.103163.

2. John L Gustafson. *The End of Error: Unum Computing*, volume 24. CRC Press, 2015.

3. John L Gustafson. Beyond Floating Point: Next Generation Computer Arithmetic. Stanford Seminar: `https://www.youtube.com/watch?v=aP0Y1uAA-2Y`, 2016. full transcription available at `http://www.johngustafson.net/pdfs/DebateTranscription.pdf`.

4. John L Gustafson. A radical approach to computation with real numbers. *Supercomputing Frontiers and Innovations*, 3(2):38–53, 2016. doi:http://dx.doi.org/10.14529/jsfi160203.

5. John L Gustafson. The Great Debate @ ARITH23. `https://www.youtube.com/watch?v=KEAKYDyUua4`, 2016. full transcription available at `http://www.johngustafson.net/pdfs/DebateTranscription.pdf`.

6. Ulrich W Kulisch and Willard L Miranker. *A new approach to scientific computation*, volume 7. Elsevier, 2014.

7. More Sites. IEEE standard for floating-point arithmetic. *IEEE Computer Society*, 2008. DOI:10.1109/IEEESTD.2008.4610935.

8. Isaac Yonemoto. `https://github.com/interplanetary-robot/SigmoidNumbers`.